DAY 3: TOWARDS SCALABLE DEEP LEARNING

Is my code Fast? Performance Analysis

2021-02-03 | Stefan Kesselheim | Helmholtz AI @ JSC

JÜLICH Forschungszentrum

OUTLINE

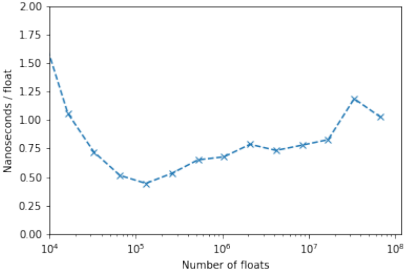
Performance of Deep Learning

Building IO Pipelines

INTRODUCTION: A SIMPLE EXAMPLE

What is the runtime of this piece of code?

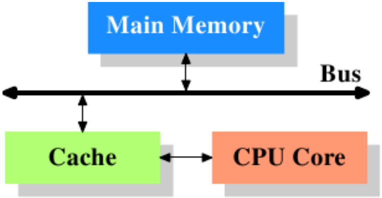
```
n=2**20                                   # For example, 1 Million Floats
m=np.random.normal(0,1,n).astype(np.float64)  # Init randomly, runtime irrelevant
mean=m.mean()                             # How long does this take?
```



- Laptop Frequency $\sim$ 2 GHz
- 1 Flop / cycle — 0.5 ns / float

MEMORY BUS

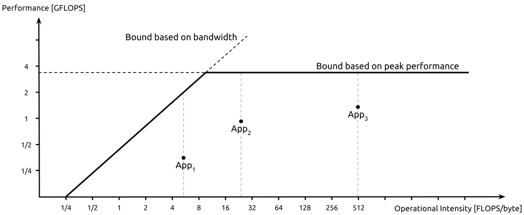
Simple architecture model



- Laptop Frequency: $\sim$ 2 GHz
- 1 Flop / cycle — 0.5 ns / float
- DDR4 Bandwidth: $\sim$ 12 GByte/sec – 0.66 ns / float
- Conclusion: Memory bandwidth is not a bottleneck single core of my laptop.
- In general, the performance can be memory-bound.

THE ROOFLINE MODEL

Arithmetic intensity: Number of Flop / Byte



ToDo:

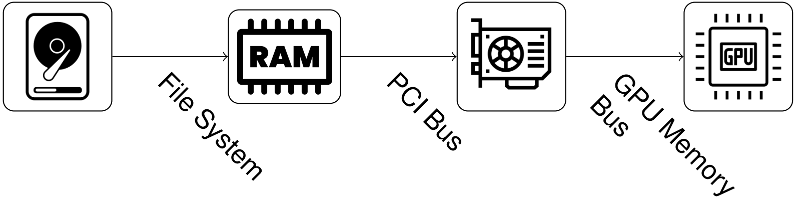
- Check your peak compute performance.
- Check you memory bandwidth.
- Determine the minimum arithmetic intensity.
- Exercise: Optimize your memory access patterns!

CONVOLUTIONAL NEURAL NETWORK

Single convolution 128x128x16, 16 channels, float32

- Input and output size: 1 MB , Weight size 2.25 kB (cached).
- Total float ops: 72 MFlop.
- Arithmetic intensity: $n_{\text{out}} \cdot k_x \cdot k_y / 4 = 36$
- Peak Compute (A100): 21 TFlop/sec (FP32)
- GPU Memory Bandwidth (A100): 1.6 TByte / sec
- Minimum arithmetic intensity 13 (FP32)
- Peak Compute (A100): 151 TFlop/sec (TP32)
- Minimum arithmetic intensity 94 (TP32)

THE BOTTLENECKS IN DL

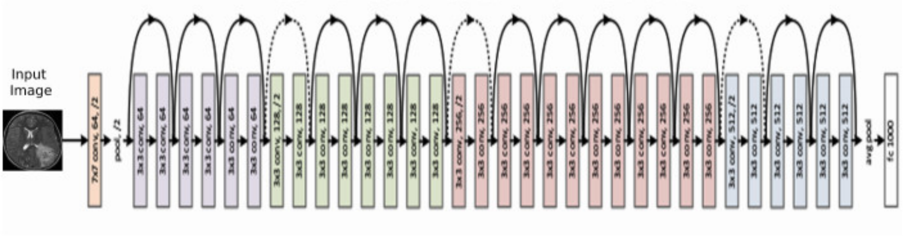


- File System Bandwidth: 10 GByte /sec (its complicated)
- PCIe 4.0x16 Bandwidth: 32 GByte / sec
- GPU-GPU Bandwidth (NVLinkv3): 600 GByte / sec
- Peak Compute (A100): 21 TFlop/sec (FP32)
- GPU Memory Bandwidth (A100): 1.6 TByte / sec

CASE ANALYSIS: RESNET50 TRAINING ON IMAGENET

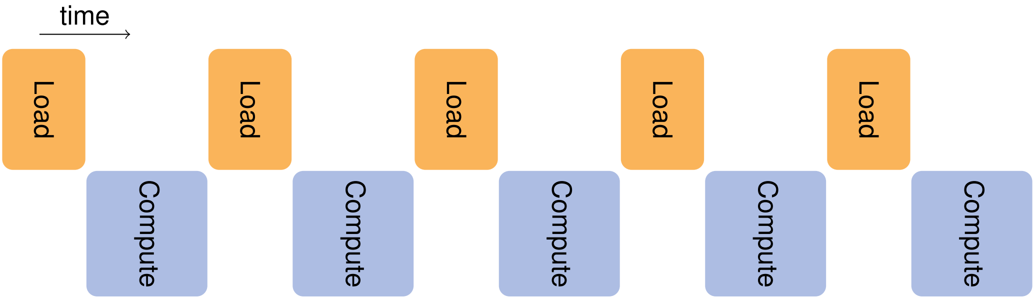
- Dataset size: 1.2 M Images, Training Resolution: 224x224x3
- Original Data: JPGs of different sizes, total 140 GB
- Uncompressed, resized to 224x224x3 data size: 180 GB
- PCIe limit 200k Images / sec.
- ResNet50 gradient computation: $\sim$ 20 GFlop.
- Compute Limit per GPU: (FP32) 1k Images / sec (TF32) 7k Images /sec
- Total weight size: 100 MB (float32)
- Dominating Operations: 3x3 Conv2D on 128x128x64, 64x64x128, 32x32x256, 16x16x512, Intensities: 144, 288, 576,1156

ResNet-50 Model Architecture

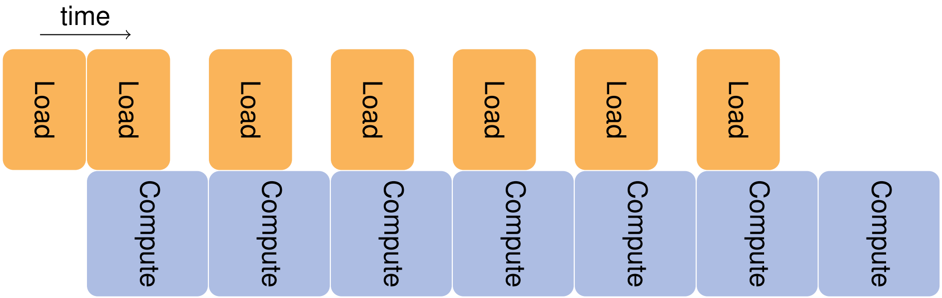


SERIAL EXECUTION

```
def load_data():
    return np.random.normal(0,1, (224,224,3)),

# Define Model
inp=tf.compat.v1.placeholder(shape=(1,224,224,3),dtype=tf.float32 )
output = tf.keras.layers.Conv2D(16, kernel_size=(3,3), use_bias=False)(inp)
# Prepare Session
sess=tf.compat.v1.Session()
sess.run(tf.compat.v1.initialize_all_variables())
# Run Model
data=load_data()
sess.run(output, feed_dict={inp: data })
```

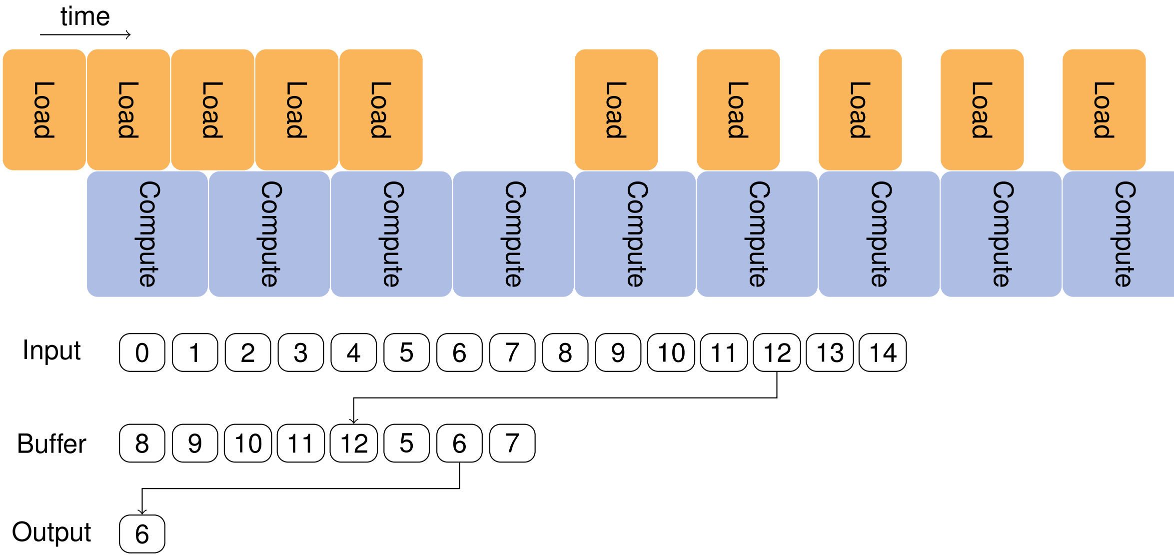


PREFETCH: ASYNCHRONOUS EXECUTION



- Parallel execution of loading and compute.
- Buffered: Load operation fills a buffer, compute consumes it.
- The buffer must be adjusted to the problem size.
- Example of latency hiding.
- Tensorflow dataset API: An easy way to do that.

PREFETCH


time
Load
Load
Load
Load
Load
Load
Load
Load
Load
Load
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Compute
Input
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Buffer
8 9 10 11 12 5 6 7
Output
6

THE DATASET API

```
In [1]:  import tensorflow as tf

In [2]:  def dataset_generator():
             def dataset_iterator():
                 for i in range(20):
                     yield "sample " + str(i)
             return dataset_iterator

In [3]:  # Example (pure python)
         gen=dataset_generator()

In [4]:  iterator=gen()

In [5]:  print(iterator.__next__())

         sample 0

In [6]:  iterator.__next__()

Out[6]:  'sample 1'

In [ ]:
```

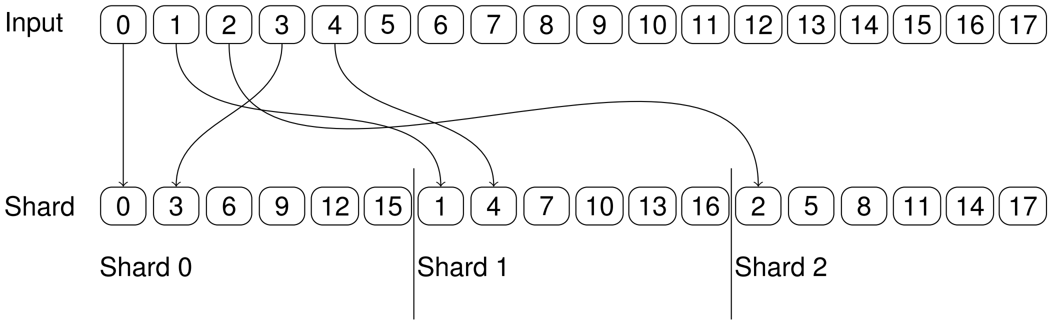
```
In [12]:  tf.compat.v1.disable_eager_execution()

In [13]:  dataset=tf.compat.v1.data.Dataset.from_generator(
              gen, output_types=tf.string)
          it=dataset.make_one_shot_iterator()
          data_tensor=it.get_next()

In [14]:  sess=tf.compat.v1.Session()
          print(sess.run(data_tensor))
          print(sess.run(data_tensor))

          b'sample 0'
          b'sample 1'
```

THE DATASET API: TF2

```
In [1]: import tensorflow as tf

In [2]: def dataset_generator():
            def dataset_iterator():
                for i in range(20):
                    tf.print("Creating Sample " + str(i))
                    yield "sample " + str(i)
            return dataset_iterator()

In [3]: dataset=tf.data.Dataset.from_generator(
            dataset_generator, output_types=tf.string)

In [4]: a=iter(dataset)

In [5]: dataset=dataset.prefetch(8)

In [6]: it=dataset.as_numpy_iterator()
        it.next()

        Creating Sample 0
        Creating Sample 1
        Creating Sample 2

Out[6]: b'sample 0'

        Creating Sample 3
        Creating Sample 4
        Creating Sample 5
        Creating Sample 6
        Creating Sample 7
        Creating Sample 8
```

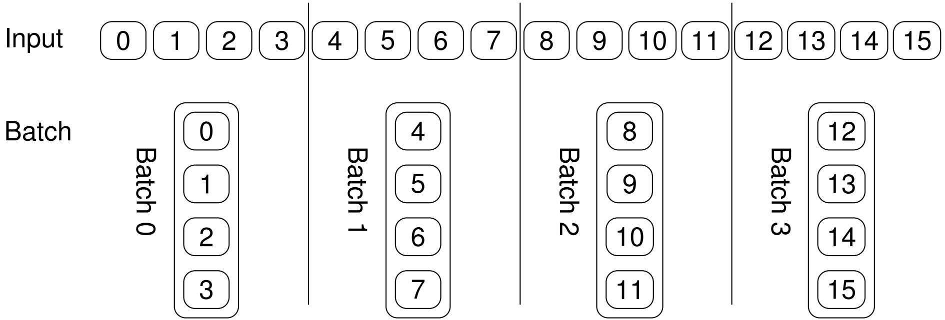
- Eager execution: The compute graph is constructed on the fly.
- `from_generator` receives a generator function, a callable that creates an iterator. So Keras can restart the iterator after each epoch.
- `Datasets` can be transformed with a functional API
- `prefetch(<num>)` creates and fills a buffer.

SHARD



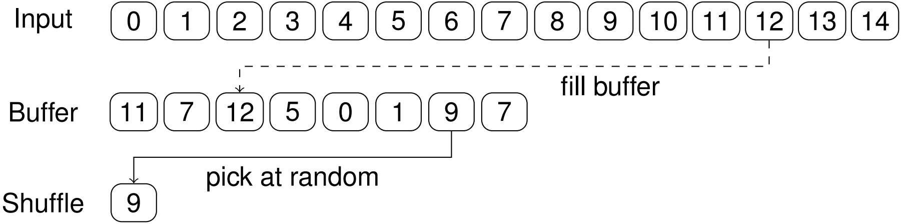
- Using `shard(i,n)` will first skip the first *i* entries in the dataset.
- Then it will skip *n* entries.
- Thus you will get only those samples with index k, where $k \bmod n = i$.
- Thus, a not can get its shard, even random access is not available.

BATCH



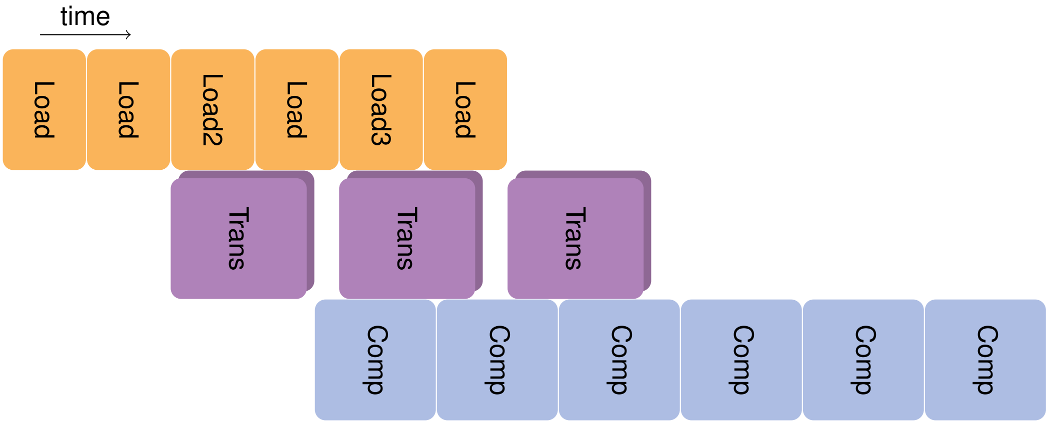
- `batch(n)` will accumulate *n* samples and return a batched tensor.
- It will only load the samples after the next item was pulled, so combine with prefetch!
- The inverse operation is `unbatch`.

SHUFFLE



- `shuffle(n)` buffer *n*.
- In each iteration, it will return a sample randomly from the buffer.
- The buffer is only refilled when needed. Combine with prefetch!
- Note that it yields only a limited randomization.

MAP



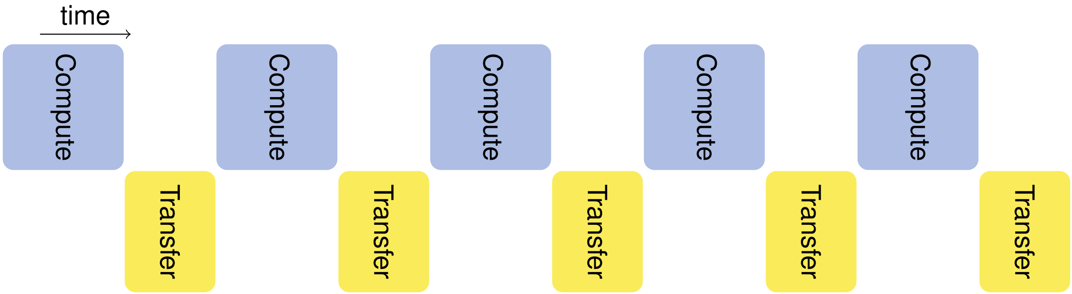
- `map(fun, n_parallel_calls` will apply a python function on each element.
- The execution is can be parallelized.
- (Pure) python and parallelization can be troublesome. Beware of the cliffs of `multiprocessing`!

GOOD PRACTICES

- Store your data with a **transparent order** on disk. Otherwise you cannot do sequential read and this may be expensive.
- Do **not** store data in **many small files**.
- Your dataset fits into the node's main memory? Easy. **Read sequentially**.
- Your dataset **does not fit** into main memory?
  - Make sure you can read you data seqentially in chunks.
  - Many relatively large files? OK.
  - File format with defined storage order and support for sequential reading? Perfect.
  - Store data pre-shuffled. Otherwise you are likely to get random-access to HD.
- Perform pre-processing on the fly, preferably directly in native tf, if necessary with parallel `map`.

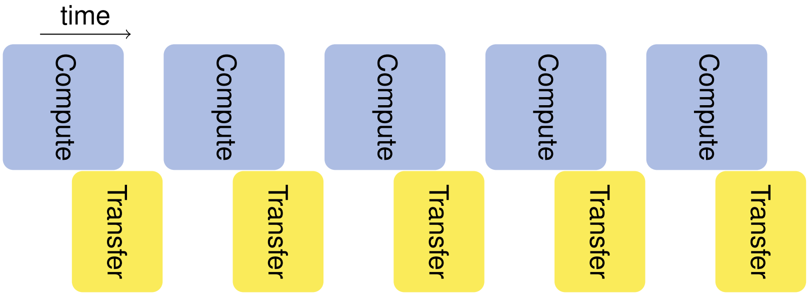
NETWORK ANALYSIS

- Infiniband Bandwidth: $\sim$ 25-50 Gigabyte/sec.
- Infiniband Latency: 150 μs
- Model size (ResNet50): 100 MB $=$ 5 ms per transfer.
- No of transfers: $\sim 2 \log_2 n_{\text{nodes}}$
- Horovod periodically checks for finished parts of the gradient. It will then start transferring if a threshold is exceeded.

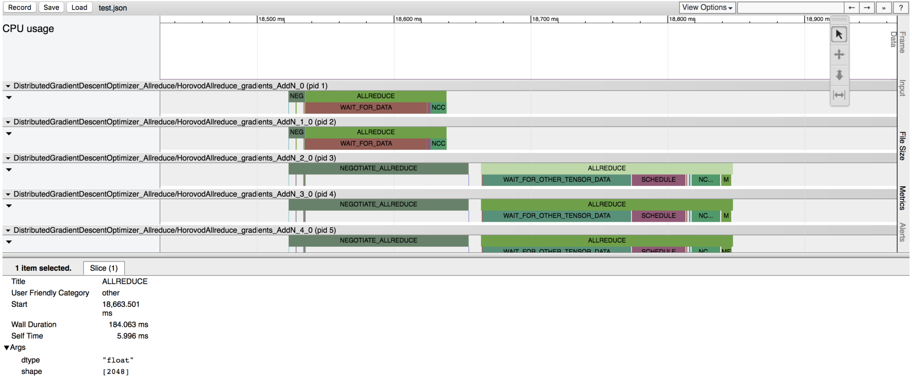


NETWORK ANALYSIS

- Infiniband Bandwidth: $\sim$ 25-50 Gigabyte/sec.
- Infiniband Latency: 150 μs
- Model size (ResNet50): 100 MB $=$ 5 ms per transfer.
- No of transfers: $\sim 2 \log_2 n_{\text{nodes}}$
- Horovod periodically checks for finished parts of the gradient. It will then start transferring if a threshold is exceeded.

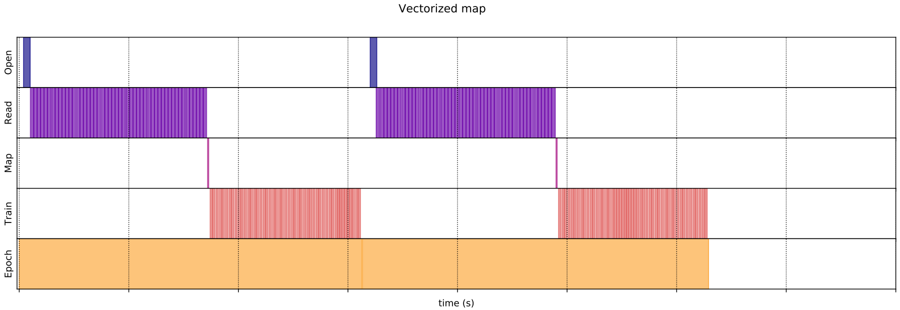


HOROVOD TIMELINE



- Horovod Timeline: Get a timeline of transfers
- Easy to use: `horovodrun -np 4 -timeline-filename /path/to/timeline.json python train.py`
- Open with chrome tracing.

TENSORBOARD PROFILER



Start with ssh tunnel in a single command on the login node:

```
ssh -L 8889:localhost:53415 kesselheim1@juwels.fz-juelich.de "bash -c \"source /p/project/
    training2004/course2021_working_environment/activate.sh && tensorboard  --port 53415 --
    logdir /p/project/training2004/kesselheim1/ \" "
```

Navigate to http://localhost:8889

Please change remote port from 53415 to you favourite random number above 1024.